Advanced Java Programming Course

Remote Method Invocation

By Võ Văn Hải
Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

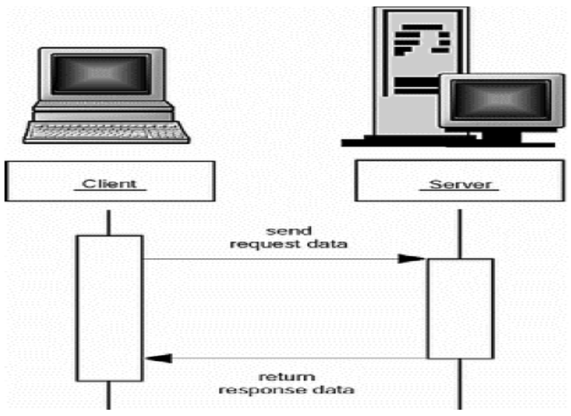
- Remote Method Invocation (RMI)

2

Remote Method Invocation

The Roles of Client and Server

synchronized request/response model

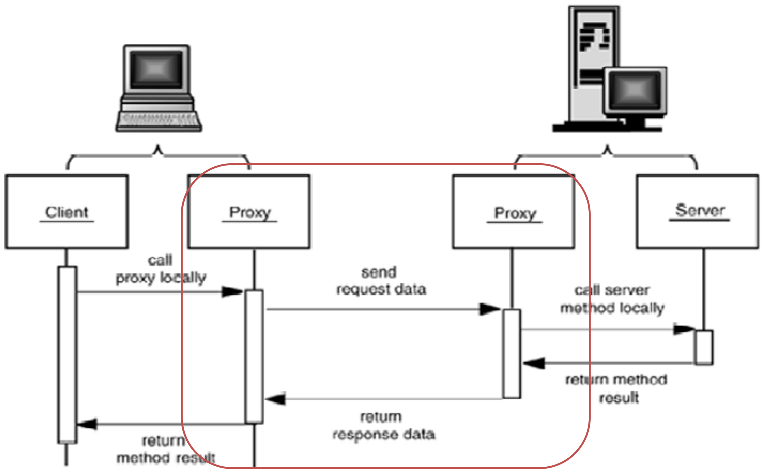


Transmitting objects between client and server

4

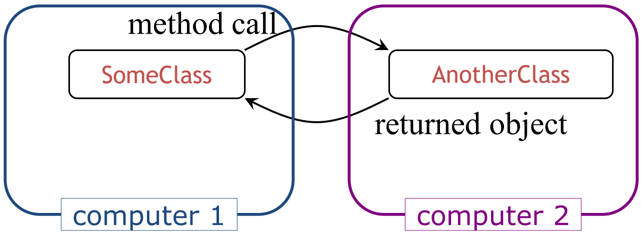
Remote Method call with proxies

new approach



"The network is the computer"*

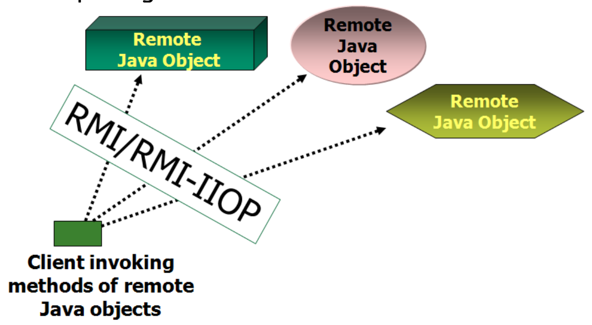
- Consider the following program organization:



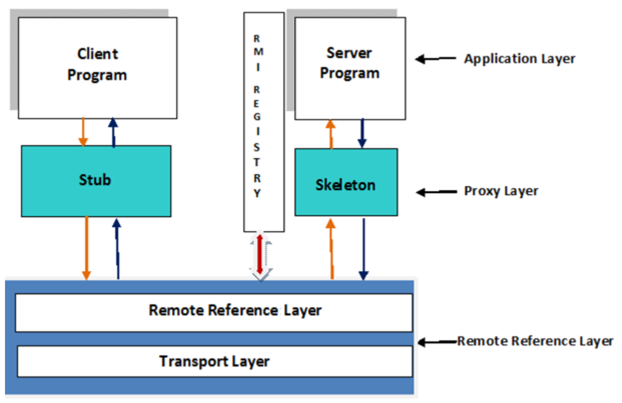
- If the network *is* the computer, we ought to be able to put the two classes on different computers
- RMI is one technology that makes this possible

What is RMI?

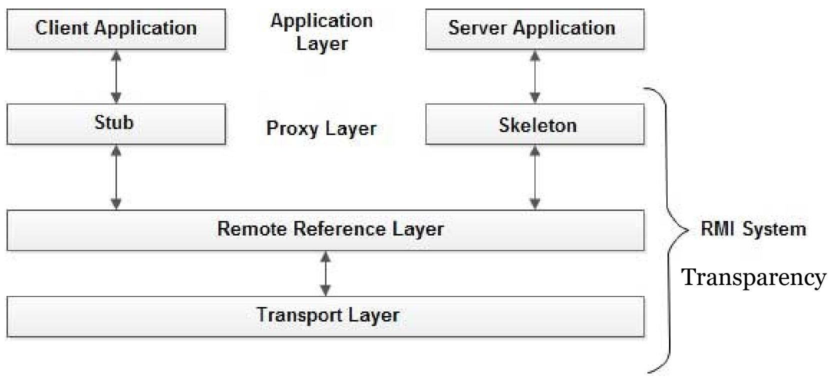
- RMI allows objects in one JVM to invoke methods of objects in another JVM.
- All in java.rmi package



The RMI architecture



The RMI architecture



Architecture of RMI

RMI Components

1. RMI Server provides an RMI service.

2. RMI Client invokes object methods of RMI service

3. "rmiregistry" program
   - Runs as a separate process
   - Allows applications to register RMI services
   - Allows obtain a reference to a named service.

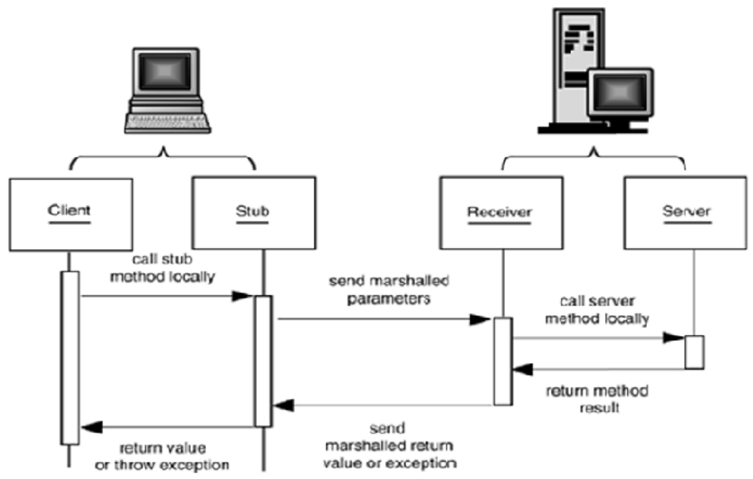
4. Stub object
   - ~~Resides on the client side~~
   - ~~Responsible for passing a message to a remote RMI service, waits for a response, and returns this response to the client.~~

Stub (client)

- When client code wants to invoke a remote method on a remote object, it actually calls an ordinary method of the Java programming language that is encapsulated in a surrogate object called a stub.
- The stub packages the parameters used in the remote method into a block of bytes.
- This packaging uses a device-independent encoding for each parameter.*(The process of encoding the parameters is called **parameter marshalling**.)*
- The purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one virtual machine to another.

Receiver(server-stub)

Parameters marshalling


Client
Stub
Receiver
Server
call stub method locally
send marshalled parameters
call server method locally
return method result
send marshalled return value or exception
return value or throw exception

13

Implementing RMI application process

1. Define RMI Service Interface *(both sides)*
2. Implement RMI Service Interface *(server side)*
3. Create RMI Server program *(server side)*
4. Create RMI Client program *(client side)*
5. Running the RMI application

14

#1/5 Define RMI Service Interface

- Your client program needs to manipulate server objects, but it doesn't actually have copies of them.
- Their capabilities are expressed in an interface that is shared between the client and server and so resides simultaneously on both machines.
- The interface characteristics:
  - Must be public
  - Must extend the interface java.rmi.Remote
  - Every method in the interface must declare that it throws java.rmi.RemoteException (other exceptions may also be thrown)

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Calc_interface extends Remote
{
    public long Add(int a,int b) throws RemoteException;
}
```

15

#2/5 Implementing RMI Service Interface

- On the server side, you must implement the class that actually carries out the methods advertised in the remote interface
- The class characteristics:
  - Should extend java.rmi.server.UnicastRemoteObject
  - Must implement a Remote interface
  - Must have a default(parametterless) constructor.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Calc_Impl extends UnicastRemoteObject implements Calc_interface
{
    public Calc_Impl()throws RemoteException{
    }
    @Override
    public long Add(int a,int b) {
        return (long)(a+b);
    }
}
```

16

#3/5 Creating RMI Server program

- The RMI server is responsible for:
  1. Instance object of a service implementation class
  2. Registering it with the RMI registry

```
import javax.naming.Context;
import javax.naming.InitialContext;
public class Calc_Server {
    public static void main(String[]args)throws Exception{
        Calc_interface obj=new Calc_Impl();
        Context ctx=new InitialContext();
        ctx.bind("rmi://localhost/calc_Server",obj);
        System.out.println("Server bound in Registry");
    }
}
```

17

#4/5 Creating RMI Client program

- The client obtains an object reference to the remote interface by making a lookup in the RMI registry
- Then invoking methods of the service interface.

```
import javax.naming.Context;
import javax.naming.InitialContext;

public class Calc_Client{
    public static void main(String[]args) throws Exception{
        Context ctx=new InitialContext();
        Calc_interface calc=
                (Calc_interface)ctx.lookup("rmi://localhost/calc_Server");
        long x=calc.Add(100,40);
        System.out.println ("result: "+x);
    }
}
```

18

#5/5 Running the RMI System

- Start Server
  - Compile all java file as normal.
    cmd: `javac *.java`
  - Start rmiregistry program.
    cmd: `start rmiregistry`
  - Run server program.
    cmd: `start java Calc_Server`
- Start Client
  - Create policy file
    cmd: `client.policy`
  - Run client
    cmd: `java -Djava.security.policy=client.policy Calc_Client`

19

Security

- By default, the RMISecurityManager restricts all code in the program from establishing network connections.
- However, the program needs to make network connections
  - To reach the RMI registry; and
  - To contact the server objects
- To allow the client to connect to the RMI registry and the server object, you supply a policy file.
- Here is a policy file that allows an application to make any network connection to a port with port number of at least 1024:

```
grant{
        permission java.net.SocketPermission
                              "*:1024-65535", "connect,listen,resolve,accept";
};
```

20

Others

- Listing all bounded remote objects in registry

```
void listingAllObjects()throws Exception{
    Context ctx = new InitialContext();
    NamingEnumeration<NameClassPair> lst=ctx.list("rmi:");
    while (lst.hasMore()){
        System.out.println(lst.next().getName());
    }
```

- **Create your own RMIRegistry**

```
java.rmi.registry.LocateRegistry.createRegistry(1099);
```

Activation object

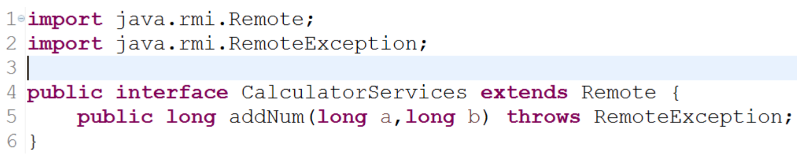
- Betty thought that if the RMI server keeps running without any connection, it seems wasting resources.
- It would be better if a remote object is delivered, shuts down itself when necessary and is activated on demand, rather than running all the time.
- Actually, Java RMI activation daemon, **rmid** is designed to do such job. The activation daemon will listen and handle the creation of activatable object on demand.

Activatable remote object

- An activatable remote object is a remote object that starts executing when its remote methods are invoked and shuts itself down when necessary.
- How to create an activatable object:
  - Create a class extends *java.rmi.activation.Activatable* class
  - and this class has a constructor to accept *ActivationID* and *MarshalledObject* parameters.

Step to develop: create activator interface

- Create activator service interface



```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CalculatorServices extends Remote {
    public long addNum(long a,long b) throws RemoteException;
}
```

Step to develop: create acivatable object

- Create activatable object

```java
import java.rmi.MarshalledObject;
import java.rmi.RemoteException;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;

public class CalcutatorImpl extends Activatable
        implements CalculatorServices{
    public CalcutatorImpl(ActivationID id, MarshalledObject<?> data)
            throws RemoteException {
        super(id, 0);
    }
    //business method
    public long addNum(long a, long b) throws RemoteException {
        return a+b;
    }

}
```

Step to develop: Create setup (server) program

- To make a remote object accessible via an activation identifier over time, you need to register an activation descriptor for the remote object and include a special constructor that the RMI system calls when it activates the activatable object.

- The following classes are involved with the activation process:
  - ActivationGroup class -- responsible for creating new instances of activatable objects in its group.
  - ActivationGroupDesc class -- contains the information necessary to create or re-create an activation group in which to activate objects in the same JVM.
  - ActivationGroupDesc.CommandEnvironment class -- allows overriding default system properties and specifying implementation-defined options for an ActivationGroup.
  - ActivationGroupID class -- identifies the group uniquely within the activation system and contains a reference to the group's activation system.
    - getSystem()-- returns an ActivationSystem interface implementation class.
  - MarshalledObject class -- a container for an object that allows that object to be passed as a paramter in an RMI call.

27

Create a set-up program

1. Install security manager for the *ActivationGroup* VM.
2. Set security policy
3. Create an instance of *ActivationGroupDesc* class
4. Register the instance and get an *ActivationGroupID*.
5. Create an instance of *ActivationDesc*.
6. Register the instance with rmid.
7. Bind or rebind the remote object instance with its name
8. Exit the system.

28

```
//step 1: Install security manager for the ActivationGroup VM.
System.setSecurityManager(new SecurityManager());
//step 2: Set security policy
Properties pros = new Properties();
pros.put("java.security.policy","rmi.policy");
//step 3: Create an instance of ActivationGroupDesc class
ActivationGroupDesc.CommandEnvironment ae = null;
ActivationGroupDesc group = new ActivationGroupDesc(pros, ae);
//step 4:Register the instance and get an ActivationGroupID.
ActivationGroupID groupId = ActivationGroup.getSystem().registerGroup(group);
//step 5:Create an instance of ActivationDesc.
String location = "file:.\\";
MarshalledObject<Object> data = null;
ActivationDesc desc = new ActivationDesc(groupId,"CalcutatorImpl",location,data);
//step 6:Register the instance with rmid.
CalculatorServices calcu = (CalculatorServices) Activatable.register(desc);
//step 7:Bind or rebind the remote object instance with its name
Naming.rebind("rmi://localhost:1099/CalculatorServices",calcu);
System.out.print("Object is hosting");
//step 8:Exit the system.
System.exit(0);
```

29

Step to develop: client

```
import java.rmi.Naming;

public class Client {
    public static void main(String[] args) throws Exception{
        Object obj=Naming.lookup(
                "rmi://localhost:1099/CalculatorServices");
        CalculatorServices cal = (CalculatorServices)obj;
        long kq = cal.addNum(1, 3);
        System.out.println("Result : "+kq);
    }
}
```

30

Step to develop: policy file

- In order to run the code successfully, we need a policy file.
- The following policy file is for all permissions.

```
grant {
    permission java.security.AllPermission;
};
```

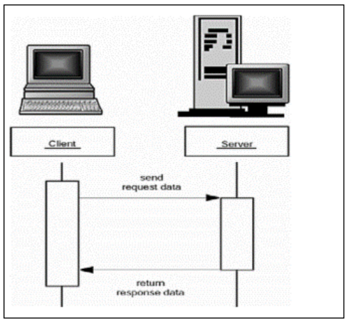
- For specific permission, you need to consult related documentation.

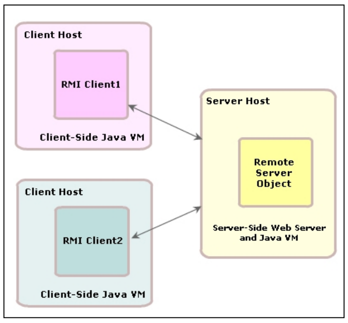
Step to develop: running

- Compile 4 classes
  - Using ***javac*** tool
- Start the rmiregistry
  - Using command: ***start rmiregistry***
- Start the activation daemon, rmid
  - Using command: ***start rmid -J-Djava.security.policy=rmi.policy***
- Run the setup program
  - Using tool: ***java -Djava.security.policy=rmi.policy*** **YourServer**
- Run the client
  - Using tool: ***java -Djava.security.policy=rmi.policy*** ***YourClient***

Summary

- Data exchange through network using Socket, UDP
- Remote method invocation





FAQ

That's all for this session!

Thank you all for your attention and patient !